□□□**Work in progress**□□□ usable, but documentation is fragmentary

# PipeStreams

PipeStreams use (pull-stream)[https://github.com/pull-stream/pull-stream]s as infrastructure to realize rather performant streaming in NodeJS. The main purpose for PipeStreams is to facilitate the building of streaming applications; in other words: to provide a simple and clear API to minimize mental overhead.

While PipeStreams as such are not directly compatible with 'classical' NodeJS push-style streams, one can always interface the two using a number of adaptors to maintain interoperability.

PipeStreams encourages and simplifies the use of classical command line (shell/bash) tools to boost performance.

## @spawn = ( command, settings ) ->

PipeStreams `spawn` is a re-imagination of how to deal with spawning child processes in an asynchronous world. Many attempts to provide for proper process spawning are either too simplistic or too hard to use right; in either case, there is too much opportunity to get things not quite right and to produce subtly faulty code that opens the door to silent failures; conversely, client code that does manage to consider all edge cases is frequently overly convoluted.

The underlying platform—NodeJS—does get some things right: it is generally a good idea to listen to streams of output from spawned processes rather than to wait for one chunk of data that only arrives when the process has finished; likewise, listening in on events is superior to evaluating exit codes on a channel that also carries result data and / or error messages. The difficulty with NodeJS `spawn`, however, is that there are so many loose ends that have to be tied together before a single process can be started: There's the child process' `stdout` and `stderr` streams that may both produce data (that may or may not be indicative of success or failure, depending on the command executed) on the one hand, and there's the `error`, `disconnect` and `exit` events, the last of which may communicate either a numerical code or else a signal name.

The basic insight that guided the implementation of `pipestreams.spawn` is threefold:

■ Fewer datasources are easier to handle than many, provided that different points of data origin remain discernible and are not just poured into one big pot; hence, output to `stdout` has to remain distinct from output to `stderr`.

■ Streams are an appropriate and manageable abstraction for the data that results from spawning an asynchronous sub-process.

■ The fewer the types of events that come down the stream and the more predictable their relative ordering, the better; especially the shape of the terminating event should be clear from the outset, because that single piece of information decides what will be communicated to the continuation and when.

PipeStreams `spawn` returns a single in the form of a pull stream source; this is the single source of 'truth' when it comes to handling success or failure (or, indeed, both, as they can co-occur in complex shell commands). That source provides the following characteristics and guarantees:

■ `spawn` is initiated with a command *string* or a command *list*; in both cases, a shell is started (`/bin/sh`).

■ Use of the shell may be explicitly disabled by passing in `{ shell: false, }` as optional 2nd argument to `PS.spawn`. Also, as with NodeJS `child_process.spawn`, a custom command may be given to replace `/bun/sh` with another executable.

■ Note that the outcome of a failing command will differ significantly depending on whether a shell was invoked or not. The difference is due to the fact that executing even a bogus command *with* a shell will generally succeed insofar as the execution of the shell *itself* is concerned (i.e. `/bin/sh` will almost never lead to an `ENOENT` condition). Once that first level of indirection has succeeded, it is the shell, not the OS that produces eventual error messages, and the way it does that is by printing to `stderr` and setting an exit code■**unhandled event: [".","entity","nbps",{"line_nr":61,"col_nr":70,"markup":""}]**■≠■**unhandled event: [".","entity","nbps",{"line_nr":61,"col_nr":70,"markup":""}]**■0. For ease-of-use it is probably best to stick to either always or else never using the shell. The reason `{ shell: true, }` has been made the default is that on modern systems it should incur only a minimal overhead while providing the more general and more flexible mechanism way to do things.

■ Shell error handling is a beast, what with poorly documented OS error codes, bash exit codes that totally differ from those even when they *mean* the same (OS: #2 `ENOENT` == Bash #127 `command not found`), signal names that are associated with numbers in that *very same small namespace* of small non-negative integers, and userland programs that exit with whatever codes they see fit (e.g. `getopt --test; echo $?` gives you `4` to indicate *success* ■**unhandled event: ["(","strike",{},{"line_nr":71,"col_nr":84,"markup":"<strike>"}]**■WTF■**unhandled event: [")","strike",null,{"line_nr"**

It's a mess. As an experimental feature, `PS.spawn` supplies an 'exit comment', that is, a short informative, standardized text to go with the `exit` event (as `event.comment`). It is currently directly derived from the exit code (and is the same whether `shell` was set or not, though that may change). You can supply your own command-specific codes; those will be looked up first when commenting on exit (e.g. use `{ comments: { 1: 'wrong port', 33: 'host unreachable', error: 'an error has occurred', } }` to add comments for exit codes 1, and 33 and set the default error comment, which itself defaults to `'error'`). When a signal has been detected and a comment has not been set otherwise, then the comment is set to the name of the signal; thus, users will be able to do a lot of error checking merely by looking at the `comment` property of the `exit` event.

■ The method returns a pull streams source. The events that come down the source will all be `[ key, value, ]` pairs (also known as 'facets').

■ Every key will be one of `'command'`, `'stdout'`, `'stderr'`, or `'exit'` (occasionally, `'error'` and `'disconnect'` may also occur, but I have yet to find trigger conditions for those).

■ The `command` event will always be the first event to come down the stream; its value is always the first argument with which `spawn` was initiated. Next come the `stdout` and `stderr` events. The last event is always `exit`; its value is an object with a `code` and a `signal` property.

■ In between the initial `command` and the final `exit` events, any number of `stdout` and `stderr` events may occur, depending on the command(s) executed. **Note that the relative ordering between output events is not well-defined across sub-commands**; when you have two commands that both write to output, then it is possible that you see the events coming in their 'natural' order *most* of the time and 'out of order' *some* of the time (relative ordering of each sub-command's writes to a single channel *will* be preserved, though). **There's no way to fix this** as it is caused by system-level contingencies.

■ The `signal` property of the `exit` event will name the signal, if any, with which the spawned process was terminated; if it is present and known and the `code` value was not also set (which should be impossible), then the `code` value is set to `128` plus the numerical equivalent (the signal number) of the signal. Otherwise, only `code` is set; in most cases, it will be `0` indicating success or else a value greater than `0` (frequently `1`) indicating failure.

■ Since it turns out that in practice neither error codes, nor signals or output to `stderr` are sufficiently reliable to judge about success or failure in a generic fashion, PipeStreams `spawn` will *never* error out with allowable inputs.

To drive that last point home, consider that calling `spawn 'bonkers'` is totally legal even if you have no executable called `bonkers` on the path; it should produce three events, in this order:

```
["command","bonkers"]
["stderr","/bin/sh: bonkers: command not found"]
["exit",{"code":127,"signal":null}]
```

spawn cannot tell whether—maybe!—what you wanted was indeed testing whether bonkers was installed on the system. At any rate, erroring out because stderr (!) and exit code (!!) would not be a wise thing to do for a generic utility, because a slight adjustment totally changes things:

```
["command","bonkers 2>■unhandled event: [".","entity","1",{"line_nr":125,"col_nr":130,"markup":"``"}]■ exit 0"]
["stdout","/bin/sh: bonkers: command not found"]
["exit",{"code":0,"signal":null}]
```

Now the error message is hidden in stdout, and the exit code looks just fine. Add to this that some executables routinely write their informational messages to stderr and may not even communicate 'errors', just 'conditions' using exit codes, and should become abundantly clear that it can only be the responsibility of the one who spawns a process to analyze the output as seen fit. This becomes especially true with compound commands; here is a sample that mixes success and failure:

```
["command","bonkers; echo "success!"; exit 0"]
["stdout","success!"]
["stderr","/bin/sh: bonkers: command not found"]
["exit",{"code":0,"signal":null}]
```

Worse is possible:

```
["command","bonkers; echo "success!"; kill -27 $$"]
["stdout","success!"]
["stderr","/bin/sh: bonkers: command not found"]
["exit",{"code":155,"signal":"SIGPROF"}]
```

## spawn **settings**

All of the settings present in the optional second argument to PS.spawn are passed through to the underlying NodeJS child_process.spawn method **except** for

■ binary <boolean>—indicates whether stdout will be left as a buffer, or else decoded as UTF-8 text and split into lines (using PS.$split()).

■ error_to_exit <boolean>—when true, will collect all lines sent to stderr and include them in the exit event value under the key error, joined with newline characters (n). If no events came over stderr or if the resulting string was empty, value.error will be set to null. This is intended to facilitate error handling in a lot of cases where exit codes and / or messages sent to stderr indicate problems with a command.

■ on_data—an event handler to be called soon as [ 'stdout', data, ] or [ 'stderr', data, ] events have transpired. This is handy especially when using spawn_collect, as you can then process e.g. progress messages from your command just-in-time and still leave exit handling to the 'main' callback function. Observe that giving an on_data event handler will cause all stdout and stderr events to be sent there, not into the stream (in the case of PS.spawn) and not to the value passed to the callback of PS.spawn_collect.

The other settings are as follows (text copied from the NodeJS docs[1]):

■ cwd <string>—Current working directory of the child process

■ env <Object>—Environment key-value pairs

■ argv0 <string>—Explicitly set the value of argv[0] sent to the child process. This will be set to command if not specified.

■ stdio <Array> | <string>—Child's stdio configuration. (See options.stdio)

■ detached <boolean>—Prepare child to run independently of its parent process. Specific behavior depends on the platform, see options.detached)

■ uid <number>—Sets the user identity of the process. (See setuid(2).)

■ gid <number>—Sets the group identity of the process. (See setgid(2).)

■ shell <boolean> | <string>—If true, runs command inside of a shell. Uses /bin/sh on UNIX, and process.env.ComSpec on Windows. A different shell can be specified as a string. See Shell Requirements and Default Windows Shell. Defaults to false (no shell) [in the case of a command string, and to true when a list for `command` is passed in].

## @$sample = ( p = 0.5, options ) ->

Given a `0 <= p <= 1`, interpret `p` as the *p*robability to *p*ick a given record and otherwise toss it, so that `$sample 1` will keep all records, `$sample 0` will toss all records, and `$sample 0.5` (the default) will toss (on average) every other record.

You can pipe several `$sample()` calls, reducing the data stream to 50% with each step. If you know your data set has, say, 1000 records, you can cut down to a random sample of 10 by piping the result of calling `$sample 1 / 1000 * 10` (or, of course, `$sample 0.01`).

Tests have shown that a data file with 3'722'578 records (which didn't even fit into memory when parsed) could be perused in a matter of seconds with `$sample 1 / 1e4`, delivering a sample of around 370 records. Because these records are randomly selected and because the process is so immensely sped up, it becomes possible to develop regular data processing as well as coping strategies for data-overload symptoms with much more ease as compared to a situation where small but realistic data sets are not available or have to be produced in an ad-hoc, non-random manner.

**Parsing CSV**: There is a slight complication when your data is in a CSV-like format: in that case, there is, with `0 < p < 1`, a certain chance that the *first* line of a file is tossed, but some subsequent lines are kept. If you start to transform the text line into objects with named values later in the pipe (which makes sense, because you will typically want to thin out largeish streams as early on as feasible), the first line kept will be mis-interpreted as a header line (which must come first in CSV files) and cause all subsequent records to become weirdly malformed. To safeguard against this, use `$sample p, headers: true` (JS: `$sample( p, { headers: true } )`) in your code.

**Predictable Samples**: Sometimes it is important to have randomly selected data where samples are constant across multiple runs:

- once you have seen that a certain record appears on the screen log, you are certain it will be in the database, so you can write a snippet to check for this specific one;
- you have implemented a new feature you want to test with an arbitrary subset of your data. You're still tweaking some parameters and want to see how those affect output and performance. A random sample that is different on each run would be a problem because the number of records and the sheer bytecount of the data may differ from run to run, so you wouldn't be sure which effects are due to which causes.

To obtain predictable samples, use `$sample p, seed: 1234` (with a non-zero number of your choice); you will then get the exact same sample whenever you re-run your piping application with the same stream and the same seed. An interesting property of the predictable sample is that—everything else being the same—a sample with a smaller `p` will always be a subset of a sample with a bigger `p` and vice versa.

# ToDo

- [ ] make (`stream-to-pull-stream`) `STPS.source`, `STPS.sink` methods public / rename `@_new_file_sink_using_stps`

## Generated Warnings

````
■unhandled event: [".","entity","nbps",{"line_nr":61,"col_nr":70,"markup":""}]■
■unhandled event: [".","entity","nbps",{"line_nr":61,"col_nr":70,"markup":""}]■
■unhandled event: ["(","strike",{},{"line_nr":71,"col_nr":84,"markup":"<strike>"}]■
■unhandled event: [")","strike",null,{"line_nr":71,"col_nr":84,"markup":"</strike>"}]■
■unhandled event: [".","entity","1",{"line_nr":125,"col_nr":130,"markup":"```"}]■
````